Intoduction

C++ Overview

What is C++?

- C++ was developed by Bjarne Stroustrup, as an extension to the C language.
- Despite being an 80s creation, C++ has been a popular programming language throughout these years.
- C++ is a cross-platform language that can be used to create high-performance applications and software systems.
- C++ is very close to the hardware making it comparatively easy for programmers to give the instructions directly to the system without any intermediary giving programmers a high level of control over system resources and memory.

Why should we learn C++/ Features of C++?

- C++ is one of the world's most popular programming languages.
- In today's operating systems, GUIs, and embedded systems, C++ is widely used.
- It is one of the most popular programming languages for its object-orientedness. C++ is an object-oriented programming language that gives a clear structure to programs and allows code to be reused, lowering development costs.
- With C++, you can develop applications or heavy games that can run on different platforms.
- As C++ is close to other programming languages such as C# and Java, which makes it easy for programmers to switch to C++ or vice versa while it is actually very easy to learn.

How is it different from C?

- The syntax of C++ is almost identical to that of C, as C++ was developed as an extension of C.
- In contrast to C, C++ supports classes and objects, while C does not.

Getting Started with C++

Requirements before you start

- To start using C++, you need two things:
  1.
     - A text editor, like Notepad, or an IDE, like VSCode to act as a platform for you to write C++ code
     - A compiler, like GCC to translate the C++ code you have written which is a high-level language into a low-level language that the computer will understand.

What is an IDE?

- IDE stands for Integrated Development Environment.
- It is nothing more than an enhanced version of a text editor that helps you write more efficient and nicer code.
- It helps to differentiate different parts of your codes with different colors and notifies you if you are missing some semicolon or bracket at some place by highlighting that area.
- A lot of IDEs are available, such as DEVC++ or Code Blocks, but we will prefer using VS Code for this tutorial series.

Installing VSCode

- Visit https://code.visualstudio.com/download

- Click on the download option as per your operating system.
- After the download is completed, open the setup and run it by saving VS Code in the default location without changing any settings.
- You will need to click the next button again and again until the installation process begins.

What is a Compiler?

- A compiler is used to run the program of a certain language which is generally high-level by converting the code into a language that is low-level that our computer could understand.
- There are a lot of compilers available, but we will proceed with teaching you to use MinGW for this course because it will fulfill all of our requirements, and also it is recommended by Microsoft itself.

Setting up the compiler

- Visit https://code.visualstudio.com/docs/languages/cpp
- Select C++ from the sidebar.
- Choose “GCC via Mingw-w64 on Windows” from the options shown there.
- Select the install sourceforge option.
- After the downloading gets completed, run the setup and choose all the default options as we did while installing VS Code.

Setting Path for Compiler

- Go to the C directory. Navigate into the Program Files. Then, open MinGW-64. Open MinGW-32. And then the bin folder. After reaching the bin, save the path or URL to the bin.
- Then go to the properties of ‘This PC’.
- Select ‘Advance System Settings’.
- Select the ‘Environment Variable’ option.
- Add the copied path to the Environment Variable.
- And now, you can visit your IDE and run your C++ programs on it. The configuration part is done.

Writing your first code in C++

Open VSCode. Here’s the simplest print statement we can start with.

```
#include <iostream>


int main()
{
    std::cout << "Hello World";
    return 0;
}
```

Copy

Output:

```
Hello World
```

Basic Structure & Syntax

Programming in C++ involves following a basic structure throughout. To understand that basic structure, the first program we learned writing in C++ will be referred to.

```
#include <iostream>


int main()
{
    std::cout << "Hello World";
    return 0;
}
```

Copy

Here's what it can be broken down to.

Pre-processor commands/ Header files

It is common for C++ programs to include many built-in elements from the standard C++ library, including classes, keywords, constants, operators, etc. It is necessary to include an appropriate header file in a program in order to use such pre-defined elements.
In the above program, #include <iostream> was the line put to include the header file iostream. The iostream library helps us to get input data and show output data. The iostream library also has many more uses and error facilities; it is not only limited to input and output.
Header file are both system defined and user defined. To know more about header files, go to the documentary here, https://en.cppreference.com/w/cpp/header.

Definition Section

Here, all the variables, or other user-defined data types are declared. These variables are used throughout the program and all the functions.

Function Declaration

- After the definition of all the other entities, here we declare all the functions a program needs. These are generally user-defined.
- Every program contains one main parent function which tells the compiler where to start the execution of the program.
- All the statements that are to be executed are written in the main function.
- Only the instructions enclosed in curly braces {} are considered for execution by the compiler.
- After all instructions in the main function have been executed, control leaves the main function and the program ends.

A C++ program is made up of different tokens combined. These tokens include:

- Keywords
- Identifiers
- Constants
- String Literal
- Symbols & Operators

1. *Keywords*

Keywords are reserved words that can not be used elsewhere in the program for naming a variable or a function. They have a specific function or task and they are solely used for that. Their functionalities are pre-defined.
One such example of a keyword could be return which is used to build return statements for functions. Other examples are auto, if, default, etc.
There is a list of reserved keywords which cannot be reused by the programmer or overloaded. One can find the list here, https://en.cppreference.com/w/cpp/keyword.

2. *Identifiers*

Identifiers are names given to variables or functions to differentiate them from one another. Their definitions are solely based on our choice but there are a few rules that we have to follow while naming identifiers. One such rule says that the name can not contain special symbols such as @, -, *, <, etc.
C++ is a case-sensitive language so an identifier containing a capital letter and another one containing a small letter in the same place will be different. For example, the three words: Code, code, and cOde can be used as three different identifiers.

3. *Constants*

Constants are very similar to a variable and they can also be of any data type. The only difference between a constant and a variable is that a constant's value never changes. We will see constants in more detail in the upcoming tutorials.

4. *String Literal*

String literals or string constants are a sequence of characters enclosed in double quotation marks. Escape sequences are also string literals.

5. *Symbols and Operators*

Symbols are special characters reserved to perform certain actions. Using them lets the compiler know what specific tasks should be performed on the given data. Several examples of symbols are arithmetical operators such as +, *, or bitwise operators such as ^, &.

C++ Comments

A comment is a human-readable text in the source code, which is ignored by the compiler. Comments can be used to insert any informative piece which a programmer does not wish to be executed. It could be either to explain a piece of code or to make it more readable. In addition, it can be used to prevent the execution of alternative code when the process of debugging is done.
Comments can be singled-lined or multi-lined.

Single Line Comments

- Single-line comments start with two forward slashes (//).
- Any information after the slashes // lying on the same line would be ignored (will not be executed) since they become unparsable.

An example of how we use a single-line comment

```
#include <iostream>


int main()
{
    // This is a single line comment
```

```
    std::cout << "Hello World";
    return 0;
}
```

Copy

Multi-line comments

- A multi-line comment starts with /* and ends with */.
- Any information between /* and */ will be ignored by the compiler.

An example of how we use a multi-line comment

```
#include <iostream>

int main()

{

  /* This is a

  multi-line

  comment */


  std::cout << "Hello World";

  return 0;

}
```

C++ Variables

Variables are containers for storing data values.
In C++, there are different types of variables.
Some of them are as follows:

- an integer variable defined with the keyword int stores integers (whole numbers), without decimals, such as 63 or -1.
- a floating point variable defined with keyword float stores floating point numbers, with decimals, such as 79.97 or -13.26.
- a character variable defined with the keyword char stores single characters, such as 'A' or 'z'. Char values are bound to be surrounded by single quotes.
- a boolean variable defined with the keyword bool stores a single value 0 or 1 for false and true respectively.

Declaration

We cannot declare a variable without specifying its data type. The data type of a variable depends on what we want to store in the variable and how much space we want it to hold. The syntax for declaring a variable is simple:

```
data_type  variable_name;
```

Copy

OR

```
data_type  variable_name = value;
```

Copy

The tutorial will go over data types later on. They will be dealt with in great detail.

Naming a Variable

There is no limit to what we can call a variable. Yet there are specific rules we must follow while naming a variable:

- A variable name in C++ can have a length of range 1 to 255 characters
- A variable name can only contain alphabets, digits, and underscores(_).
- A variable cannot start with a digit.
- A variable cannot include any white space in its name.
- Variable names are case sensitive
- The name should not be a reserved keyword or any special character.

Variable Scope

The scope of a variable is the region in a program where the existence of that variable is valid. Based on its scope, variables can be classified into two types:

Local variables:

Local variables are declared inside the braces of any function and can be assessed only from that particular function.

Global variables:

Global variables are declared outside of any function and can be accessed from anywhere.

An example that demonstrates the difference in applications of a local and a global variable is given below.

```
#include <iostream>

using namespace std;


int a = 5; //global variable


void func()

{

   cout << a << endl;

}


int main()

{

   int a = 10; //local variable

   cout << a << endl;

   func();

   return 0;

}
```

Output

10

5

Explanation: A local variable a was declared in the main function, and when printed, gave 10. This is because, within the body of a function, a local variable takes precedence over a global variable with the same name. But since there was no variable declared in the func function, it considered the global variable a for printing, and hence the value 5.

A variable, as its name is defined, can be altered, or its value can be changed, but the same is not true for its type. If a variable is of integer type, it will only store an integer value through a program. We cannot assign a character type value to an integer variable. We cannot even store a decimal value into an integer variable.

C++ Data Types & Constants

C++ Data Types

Data types define the type of data a variable can hold; for example, an integer variable can hold integer data, a character can hold character data, etc.
Data types in C++ are categorized into three groups:

Built-in data types

These data types are pre-defined for a language and could be used directly by the programmer.
Examples are: Int, Float, Char, Double, Boolean

User-defined data types

These data types are defined by the user itself.
Examples are: Class, Struct, Union, Enum

Derived data types

These data types are derived from the primitive built-in data types.
Examples are: Array, Pointer, Function

Some of the popular built-in data types and their applications are:

Data Type	Size	Description
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. They require 4 bytes of memory space.
double	8 bytes	Stores fractional numbers, containing one or more decimals. They require 4 bytes of memory space.
char	1 byte	Stores a single character/letter/number, or ASCII values
boolean	1 byte	Stores true or false values

C++ Constants

Constants are unchangeable; when a constant variable is initialized in a program, its value cannot be changed afterwards.

```
#include <iostream>

using namespace std;
```

```
int main()
{
    const float PI = 3.14;
    cout << "The value of PI is " << PI << endl;
    PI = 3.00; //error, since changing a const variable is not allowed.
}
```

Output:

```
error: assignment of read-only variable 'PI'
```

C++ Operators

Special symbols that are used to perform actions or operations are known as operators. They could be both unary or binary.
For example, the symbol + is used to perform addition in C++ when put in between two numbers, so it is a binary operator. There are different types of operators. They are as follows:

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, etc. They could be both binary and unary. A few of the simple arithmetic operators are

Operation	Description
a + b	Adds a and b
a - b	Subtracts b from a
a * b	Multiplies a and b
a / b	Divides a by b
a % b	Modulus of a and b
a++	Post increments a by 1
a--	Post decrements a by 1
++a	Pre increments a by 1
--a	Pre decrements a by 1

Let's see their implementation in C++.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 4, b = 5;
    cout << "The value of a + b is " << a + b << endl;
    cout << "The value of a - b is " << a - b << endl;
    cout << "The value of a * b is " << a * b << endl;
    cout << "The value of a / b is " << a / b << endl;
```

```
    cout << "The value of a % b is " << a % b << endl;
    cout << "The value of a++ is " << a++ << endl;
    cout << "The value of a-- is " << a-- << endl;
    cout << "The value of ++a is " << ++a << endl;
    cout << "The value of --a is " << --a << endl;
}
```

Copy

Output:

```
The value of a + b is 9
The value of a - b is -1
The value of a * b is 20
The value of a / b is 0
The value of a % b is 4
The value of a++ is 4
The value of a-- is 5
The value of ++a is 5
The value of --a is 4
```

Relational Operators

Relational operators are used to check the relationship between two operands and to compare two or more numbers or even expressions in cases. The return type of a relational operator is a Boolean that is, either True or False (1 or 0).

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Is equal to
!=	Is not equal to

Let's see their implementation in C++.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 4, b = 5;
    cout << "The value of a == b is " << (a == b) << endl;
    cout << "The value of a < b is " << (a < b) << endl;
    cout << "The value of a > b is " << (a > b) << endl;
}
```

Output:

```
The value of a==b is 0
The value of a<b is 1
The value of a>b is 0
```

The output is 0 for a==b, since a and b are not equal and 1 for a<b, since a is less than b.

Logical Operators

Logical Operators are used to check whether an expression is true or false. There are three logical operators i.e. AND, OR, and NOT. They can be used to compare Boolean values but are mostly used to compare expressions to see whether they are satisfying or not.

- AND: it returns true when both operands are true or 1.
- OR: it returns true when either operand is true or 1.
- NOT: it is used to reverse the logical state of the operand and is true when the operand is false.

Operator	Description
&&	AND Operator
||	OR Operator
!	NOT Operator

Let's see their implementation in C++.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 1, b = 0;
    cout << "The value of a && b is " << (a && b) << endl;
    cout << "The value of a || b is " << (a || b) << endl;
    cout << "The value of !a is " << (!a) << endl;
}
```

Output:

```
The value of a && b is 0
The value of a || b is 1
The value of !a is 0
```

Bitwise Operators

A bitwise operator is used to perform operations at the bit level. To obtain the results, they convert our input values into binary format and then process them using whatever operator they are being used with.

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

~	Bitwise Complement
>>	Shift Right Operator
<<	Shift Left Operator

Let's see their implementation in C++.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 13; //1101
    int b = 5;  //101
    cout << "The value of a & b is " << (a & b) << endl;
    cout << "The value of a | b is " << (a | b) << endl;
    cout << "The value of a ^ b is " << (a ^ b) << endl;
    cout << "The value of ~a is " << (~a) << endl;
    cout << "The value of a >> 2 is " << (a >> 2) << endl;
    cout << "The value of a << 2 is " << (a << 2) << endl;
}
```

Output:

```
The value of a & b is 5
The value of a | b is 13
The value of a ^ b is 8
The value of ~a is -14
The value of a >> 2 is 3
The value of a << 2 is 52
```

Assignment Operators

Assignment operators are used to assign values. We will use them in almost every program we develop.

```
int a = 0;
int b = 1;
```

Equal to (=) is the assignment operator here. It is assigning 0 to a and 1 to b in the above example.

Operator	Description
=	It assigns the right side operand value to the left side operand.
+=	It adds the right operand to the left operand and assigns the result to the left operand.
-=	It subtracts the right operand from the left operand and assigns the result to the left operand.
*=	It multiplies the right operand with the left operand and assigns the result to the left operand.

/=	It divides the left operand with the right operand and assigns the result to the left operand.

Operator Precedence and Associativity

Operator precedence

It helps us determine the precedence of an operator over another while solving an expression. Consider an expression a+b*c. Now, since the multiplication operator's precedence is higher than the precedence of the addition operator, multiplication between a and b is done first and then the addition operation will be performed.

Operator associativity

It helps us to solve an expression; when two or more operators having the same precedence come together in an expression. It helps us decide whether we should start solving the expression containing operators of the same precedence from left to right or from right to left.
The table containing the operator precedence and operator associativity of all operators can be found here. C++ Operator Precedence - cppreference.com.

C++ Manipulators

In C++ programming, language manipulators are used in the formatting of output. These are helpful in modifying the input and the output stream. They make use of the insertion and extraction operators to modify the output.
Here's a list of a few manipulators:

Manipulators	Description
endl	It is used to enter a new line with a flush.
setw(a)	It is used to specify the width of the output.
setprecision(a)	It is used to set the precision of floating-point values.
setbase(a)	It is used to set the base value of a numerical number.

Let's see their implementation in C++. Note that we use the header file *iomanip* for some of the manipulators.

```
#include <iostream>
#include <iomanip>
using namespace std;


int main()
{
    float PI = 3.14;
    int num = 100;
    cout << "Entering a new line." << endl;
    cout << setw(10) << "Output" << endl;
    cout << setprecision(10) << PI << endl;
    cout << setbase(16) << num << endl; //sets base to 16
}
```

Output:

```
Entering a new line.
    Output
3.140000105
64
```

C++ Basic Input/Output

C++ language comes with different libraries, which help us in performing input/output operations. In C++, sequences of bytes corresponding to input and output are commonly known as streams. There are two types of streams.
They are,

Input stream

In the input stream, the direction of the flow of bytes occurs from the input device (for ex keyboard) to the main memory.

Output stream

In the output stream, the direction of flow of bytes occurs from the main memory to the output device (for ex-display)

An example that demonstrates how input and output are popularly done in C++.

```cpp
#include <iostream>
using namespace std;

int main()
{
    int num;
    cout << "Enter a number: ";
    cin >> num;                         // Getting input from the user
    cout << "Your number is: " << num; // Displaying the input value
    return 0;
}
```

Input:

```
Enter a number: 10
```

Output:

```
Your number is: 10
```

Important Points

- The sign << is called the insertion operator.
- The sign >> is called the extraction operator.
- cout keyword is used to print.
- cin keyword is used to take input at run time.

Control Structure

The work of control structures is to give flow and logic to a program. There are three types of basic control structures in C++.

Sequence Structure

Sequence structure refers to the sequence in which program execute instructions one after another.

Selection Structure

Selection structure refers to the execution of instruction according to the selected condition, which can be either true or false. There are two ways to implement selection structures. They are done either by if-else statements or by switch case statements.

Loop Structure

Loop structure refers to the execution of an instruction in a loop until the condition gets false.

C++ If Else

If else statements are used to implement a selection structure. Like any other programming language, C++ also uses the if keyword to implement the decision control instruction.

The condition for the if statement is always enclosed within a pair of parentheses. If the condition is true, then the set of statements following the if statement will execute. And if the condition evaluates to false, then the statement will not execute, instead, the program skips that enclosed part of the code.

An expression in if statements are defined using relational operators. The statement written in an if block will execute when the expression following if evaluates to true. But when the if block is followed by an else block, then when the condition written in the if block turns to be false, the set of statements in the else block will execute.

Following is the syntax of if-else statements:

```
if ( condition ){
statements;}
 else {
statements;}
```

Copy

One example where we could use the if-else statement is:

```
#include <iostream>
using namespace std;

int main()
{
    int age;
    cout << "Enter a number: ";
    cin >> age;
    if (age >= 50)
```

```
    {
        cout << "Input number is greater than 50!" << endl;
    }
    else if (age == 50)
    {
        cout << "Input number is equal to 50!" << endl;
    }
    else
    {
        cout << "Input number is less than 50!" << endl;
    }
}
```

Input

```
Enter a number: 51
```

Output

```
Input number is greater than 50!
```

Note: The *else if* statement checks for a different condition if the conditions checked above it evaluate to false.

C++ Switch Case

The control statement that allows us to make a decision effectively from the number of choices is called a switch, or a switch case-default since these three keywords go together to make up the control statement.
Switch executes that block of code, which matches the case value. If the value does not match with any of the cases, then the default block is executed.

Following is the syntax of switch case-default statements:

```
switch ( integer/character expression )
{
case {value 1} :
do this ;

case {value 2} :
 do this ;

default :
do this ;
 }
```

The expression following the switch can be an integer expression or a character expression. Remember, that case labels should be unique for each of the cases. If it is the same, it may create a problem while executing a program. At the end of the case labels, we always use a

colon (:). Each case is associated with a block. A block contains multiple statements that are grouped together for a particular case.

The break keyword in a case block indicates the end of a particular case. If we do not put the break in each case, then even though the specific case is executed, the switch will continue to execute all the cases until the end is reached. The default case is optional. Whenever the expression's value is not matched with any of the cases inside the switch, then the default case will be executed.

One example where we could use the switch case statement is

```
#include <iostream>
using namespace std;

int main()
{
    int i = 2;
    switch (i)
    {
    case 1:
        cout << "Statement 1" << endl;
        break;

    case 2:
        cout << "Statement 2" << endl;
        break;

    default:
        cout << "Default statement!" << endl;
    }
}
```

Output

```
Statement 2
```

The test expression of a switch statement must necessarily be of an integer or character type and the value of the case should be an integer or character as well. Cases should only be inside the switch statement and using the break keyword in the switch statement is not necessary.

C++ Loops

The need to perform an action, again and again, with little or no variations in the details each time they are executed is met by a mechanism known as a loop. This involves repeating some code in the program, either a specified number of times or until a particular condition is satisfied. Loop-controlled instructions are used to perform this repetitive operation efficiently ensuring the program doesn't look redundant at the same time due to the repetitions.

Following are the three types of loops in C++ programming.

- For Loop
- While Loop
- Do While Loop

For Loop

A for loop is a repetition control structure that allows us to efficiently write a loop that will execute a specific number of times. The for-loop statement is very specialized. We use a for loop when we already know the number of iterations of that particular piece of code we wish to execute. Although, when we do not know about the number of iterations, we use a while loop which is discussed next.

Here is the syntax of a for loop in C++ programming.

```
for (initialise counter; test counter; increment / decrement counter)
{
    //set of statements
}
```

Copy

Here,

- **initialize counter:** It will initialize the loop counter value. It is usually i=0.
- **test counter:** This is the test condition, which if found true, the loop continues, otherwise terminates.
- **Increment/decrement counter:** Incrementing or decrementing the counter.
- **Set of statements:** This is the body or the executable part of the for loop or the set of statements that has to repeat itself.

One such example to demonstrate how a for loop works is

```
#include <iostream>
using namespace std;

int main()
{
    int num = 10;
    int i;
    for (i = 0; i < num; i++)
    {
        cout << i << " ";
    }
    return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

First, the initialization expression will initialize loop variables. The expression i=0 executes once when the loop starts. Then the condition i < num is checked. If the condition is true, then

the statements inside the body of the loop are executed. After the statements inside the body are executed, the control of the program is transferred to the increment of the variable i by 1. The expression i++ modifies the loop variables. Iteratively, the condition i < num is evaluated again.

The for loop terminates when i finally becomes greater than num, therefore, making the condition i<num false.

While Loop

A While loop is also called a pre-tested loop. A while loop allows a piece of code in a program to be executed multiple times, depending upon a given test condition which evaluates to either true or false. The while loop is mostly used in cases where the number of iterations is not known. If the number of iterations is known, then we could also use a for loop as mentioned previously.

Following is the syntax for using a while loop.

```
while (condition test)
{
    // Set of statements
}
```

Copy

The body of a while loop can contain a single statement or a block of statements. The test condition may be any expression that should evaluate as either true or false. The loop iterates while the test condition evaluates to true. When the condition becomes false, it terminates.

One such example to demonstrate how a while loop works is

```
#include <iostream>
using namespace std;

int main()
{
    int i = 5;
    while (i < 10)
    {
        cout << i << " ";
        i++;
    }

    return 0;
}
```

Output

```
5 6 7 8 9
```

Do While Loop

A do-while loop is a little different from a normal while loop. A do-while loop, unlike what happens in a while loop, executes the statements inside the body of the loop before checking the test condition.

So even if a condition is false in the first place, the do-while loop would have already run once. A do-while loop is very much similar to a while loop, except for the fact that it is guaranteed to execute the body at least once.

Unlike for and while loops, which test the loop condition first, then execute the code written inside the body of the loop, the do-while loop checks its condition at the end of the loop.

Following is the syntax for using a do-while loop.

```
do
{
    statements;
} while (test condition);
```

Copy

First, the body of the do-while loop is executed once. Only then, the test condition is evaluated. If the test condition returns true, the set of instructions inside the body of the loop is executed again, and the test condition is evaluated. The same process goes on until the test condition becomes false. If the test condition returns false, then the loop terminates.

One such example to demonstrate how a do-while loop works is

```
#include <iostream>
using namespace std;

int main()
{
    int i = 5;
    do
    {
        cout << i << " ";
        i++;
    } while (i < 5);

    return 0;
}
```

Output

```
5
```

Here, even if i was less than 5 from the very beginning, the do-while let the print statement execute once, and then terminated.

Break Statement

Break statement is used to break the loop or switch case statements execution and brings the control to the next block of code after that particular loop or switch case it was used in.

Break statements are used to bring the program control out of the loop it was encountered in. The break statement is used inside loops or switch statements in C++ language.

One such example to demonstrate how a break statement works is

```
#include <iostream>
using namespace std;

int main()
{
    int num = 10;
    int i;
    for (i = 0; i < num; i++)
    {
        if (i == 6)
        {
            break;
        }
        cout << i << " ";
    }

    return 0;
}
```

Output

```
0 1 2 3 4 5
```

Here, when i became 6, the break statement got executed and the program came out of the for loop.

Continue Statement

The continue statement is used inside loops in C++ language. When a continue statement is encountered inside the loop, the control jumps to the beginning of the loop for the next iteration, skipping the execution of statements inside the body of the loop after the continue statement.

It is used to bring the control to the next iteration of the loop. Typically, the continue statement skips some code inside the loop and lets the program move on with the next iteration. It is mainly used for a condition so that we can skip some lines of code for a particular condition.

It forces the next iteration to follow in the loop unlike a break statement, which terminates the loop itself the moment it is encountered.

One such example to demonstrate how a continue statement works is

```
#include <iostream>
using namespace std;

int main()
{
    for (int i = 0; i <= 10; i++)
    {
        if (i < 6)
        {
            continue;
        }
        cout << i << " ";
    }
    return 0;
}
```

Output

```
6 7 8 9 10
```

Here, the continue statement was continuously executing while i remained less than 5. For all the other values of i, we got the print statement working.

Array Basics

An array is a collection of items that are of the data type stored in contiguous memory locations. And it is also known as a subscript variable.
It can even store the collection of derived data types such as pointers, structures, etc.

An array can be of any dimension. The C++ Language places no limits on the number of dimensions in an array. This means we can create arrays of any number of dimensions. It could be a 2D array or a 3D array or more.

Advantages of Arrays?

- It is used to represent multiple data items of the same type by using only a single name.
- Accessing any random item at any random position in a given array is very fast in an array.
- There is no case of memory shortage or overflow in the case of arrays since the size is fixed and elements are stored in contiguous memory locations.

Array Operations

Defining an array

1. *Without specifying the size of the array:*

```
int arr[] = {1, 2, 3};
```

Here, we can leave the square brackets empty, although the array cannot be left empty in this case. It must have elements in it.

2. *With specifying the size of the array:*

```
3. int arr[3];
```

```
arr[0] = 1, arr[1] = 2, arr[2] = 3;
```

Accessing an array element

An element in an array can easily be accessed through its index number.
An index number is a special type of number which allows us to access variables of arrays. Index number provides a method to access each element of an array in a program. This must be remembered that the index number starts from 0 and not one.
Example:

```
#include <iostream>
using namespace std;

int main()
{
    int arr[] = {1, 2, 3};
    cout << arr[1] << endl;
}
```

Output:

```
2
```

Changing an array element

An element in an array can be overwritten using its index number.
Example:

```
#include <iostream>
using namespace std;

int main()
{
    int arr[] = {1, 2, 3};
    arr[2] = 8; //changing the element on index 2
    cout << arr[2] << endl;
}
```

Output:

```
8
```